MOBILE OBJECTS "MUST" MOVE SAFELY

Sébastien Briais
ENS Lyon, France
Sebastien.Briais@ens-lyon.fr

Uwe Nestmann
EPFL, Switzerland
Uwe.Nestmann@epfl.ch

Abstract

Øjeblik is a lexically-scoped, object-based calculus that represents a distribution-free subset of the LAN-based programming language Obliq. The *surrogate* operation on Øjeblik-objects, which is the abstraction of *migration* on Obliq-objects, is a combined operation derived from the more primitive operations *cloning* and *aliasing*. In short, surrogation on an object turns the object into an alias for a clone of itself; it amounts to migration when the original and the clone reside on different distribution sites.

In previous work, we studied the conditions under which surrogation is safe, i.e., transparent to object clients. To this aim, we developed two complementary formal descriptions of Øjeblik's semantics, one as an operational semantics on Øjeblik-configurations, and another one by translation into a process calculus. We used the former to explain typical (mis-)behaviors of Øjeblik programs, but only the latter to perform rigorous correctness proofs w.r.t. may-equivalence.

In this paper, we offer new formal proofs, now based on the operational semantics of Øjeblik, making the results as well as the proofs accessible also to readers not familiar with process calculi. Furthermore, we strengthen our former results by using, in addition to may-equivalence, the much more distinguishing notion of must-equivalence.

1. Introduction

This paper addresses, like previous works [NHKM01, MKN00], the problem of expressing the mobility of objects in lexically-scoped languages like Obliq [Car95] by means of cloning and aliasing. In this sense, it is to be seen as a natural continuation of these works.

The title of this paper is intended to emphasize two different messages. Firstly, it stresses the obligation that mobile objects should indeed move in a safe way, which means that they—while moving—must not be disturbed by any other concurrent activity and that they should move without allowing their clients to take notice of it. Secondly, it hints at one of the two main new contributions of this paper, namely the fact that clients cannot observe the difference between the case in which an object has moved and the case in which it has not (yet) moved, even not up to "must-equivalence".

Relevance of the problem. In order to protect objects during migration and the resulting proxies afterwards, Obliq proposes a blocking strategy (based on serialization and protection against external modification). This strategy appears to be necessary for the proposal of mobile objects through cloning and aliasing. In such settings, the transparency-of-migration problems arise inevitably, because the blocking strategy also affects the generated proxies. Thus, our study is not only addressing Obliq, but any language that supports a blocking strategy for transparent object migration using proxies.

Previous Work. We have studied in great detail the problems of developing and exploiting formal semantics of languages arising from Obliq. In [NHKM01], guided by an implementation of Obliq, we studied four different operational semantics and formalized safe migration as the following theorem: in $x.\mathsf{ping} \cong x.\mathsf{surrogate}$ we equate (with respect to a large class of program contexts) the program $x.\mathsf{ping}$, which just witnesses the responsiveness of x, with the program $x.\mathsf{surrogate}$, which performs a surrogation operation on x. We then ruled out three of the operational semantics due to problems in satisfying the theorem, but we were not able (yet) to formally prove that our favorite semantics would indeed satisfy it. In [MKN00], we then proved the theorem to hold in our favorite semantics, but only when formalized as a translation into a suitable π-calculus [Mer00]. Furthermore, due to the character of the standard proof techniques of the π-calculus—some form of weak bisimulation, which is usually insensitive w.r.t. divergence—we only gave a proof for the safety theorem using the notion of may-equivalence $\cong^{\mathrm{may}}$, in which two terms to compare must exhibit the same may-convergence behavior in all program contexts.

Contribution. This paper provides the missing link between [NHKM01] and [MKN00]: several previous readers were missing a formal relation between the operational and the translational semantics just for completing the understanding of the problem, others were arguing that proofs on translation would be useless without such a link. Here, instead of establishing a formal correspondence, we lift some proof ideas from the level of a process calculus to the level of the operational semantics, we develop further proof techniques

(partial confluence, path compression) that enable a deeper understanding of the migration problem, and we strengthen previous results using the more distinguishing notion of must-equivalence $\cong^{\mathrm{must}}$. Our new proof techniques will be reusable for other verification tasks, as well.

Outline. § 2 recalls the necessary syntactic and semantic details of the calculus Øjeblik, our basic vehicle to study Obliq. In § 3, we briefly set up the safety theorem that we are interested in. Finally, § 4 is dedicated to summarize the highlights of a formal proof of the safety theorem using the operational semantics and must-equivalence. Full proofs are found in [Bri01].

2. Concurrent Objects with Cloning and Aliasing

Øjeblik is a typed calculus [NHKM01], but we omit types throughout this paper to keep the presentation simple. In comparison with Obliq [Car95], which is a fully-fledged LAN-based programming language, we omit ground values, data operations, and procedures, we restrict field selection to method invocation, we restrict multiple cloning to single cloning, we omit flexibility of object attributes, we replace field aliasing with object aliasing, we omit explicit distribution, and we omit exceptions and advanced synchronization, so that we get a feasible, but still non-trivial language.

2.1. Syntax

The set $\mathcal{L}$ of Øjeblik-terms is generated as shown in Figure 1, where *method labels* l and *variables* s, x, y, z are taken from countably infinite sets $\mathbf{L}$ and $\mathbf{X}$, respectively. The remainder of this subsection presents an informal explanation of the semantics of Øjeblik terms. Computation follows the call-by-value evaluation order; its goal is to reduce terms to values, which are run-time entities that we also call references (cf. Subsection 2.2 for the precise meaning).

Objects. An object record $[\mathrm{l}_j{=}m_j]_{j\in J}$ is a finite collection of updatable named methods $\mathrm{l}_j{=}m_j$, more generally called fields, for pairwise distinct labels l_j. In a method $\varsigma(s,\tilde{x})b$, the letter ς denotes a binder for the self variable s and argument variables $\tilde{x}$ within the body b. Moreover, every object in Øjeblik comes equipped with special methods for cloning, aliasing, surrogation, and ping, which cannot be overwritten by the update operation.

Method invocation $a.\mathrm{l}\langle\,\tilde{c}\,\rangle$ with field l of the object a containing the method $\varsigma(s,\tilde{x})b$ results in the body b with the self variable s replaced by (a reference to) the enclosing object a, and the formal parameters $\tilde{x}$ replaced by (references to) the actual parameters $\tilde{c}$ of the invocation. Method update $a.\mathrm{l}{\Leftarrow}m$ overwrites the current content of the named field l in object a with method m and evaluates to the modified object. The operation $a.\mathsf{clone}$ creates an object with the same

$$
\begin{array}{rll}
a, b, c & ::= [\mathrm{l}_j{=}m_j]_{j\in J} & \text{object record} \\
& \mid a.\mathrm{l}\langle\, \tilde{c}\,\rangle & \text{method invocation} \\
& \mid a.\mathrm{l}{\Leftarrow}m & \text{method update} \\
& \mid a.\mathsf{clone} & \text{shallow copy} \\
& \mid a.\mathsf{alias}\langle b\rangle & \text{object aliasing} \\
& \mid a.\mathsf{surrogate} & \text{object surrogation} \\
& \mid a.\mathsf{ping} & \text{object identity} \\
& \mid s, x, y, z & \text{variables} \\
& \mid \mathsf{let}\, x = a\, \mathsf{in}\, b & \text{local definition} \\
& \mid \mathsf{fork}\langle a\rangle & \text{thread creation} \\
& \mid \mathsf{join}\langle a\rangle & \text{thread destruction} \\
m_j & ::= \varsigma(s_j, \tilde{x}_j)b_j & \text{method}
\end{array}
$$

Figure 1. Syntax of Øjeblik expressions

fields as the original object and initializes the fields to the same entries as in the original object. The operation $a.\mathsf{alias}\langle b\rangle$ replaces object a with an alias to b, written $a{\gg}b$, regardless of whether a is already an alias; if b itself is an alias, e.g. $b{\gg}c$, then we consequently and naturally create an alias chain $a{\gg}b{\gg}c$. After the operation $a.\mathsf{alias}\langle b\rangle$, requests arriving at a are forwarded to b. The operation $a.\mathsf{surrogate}$ is the abstraction of migration: by calling it, object a is turned into an alias to a clone of itself, which is implemented by providing a uniform method $\mathsf{surrogate}{=}\varsigma(s)s.\mathsf{alias}\langle s.\mathsf{clone}\rangle$. Like standard methods, surrogation is forwarded by aliased objects. The operation $a.\mathsf{ping}$ is also implemented by providing a uniform method: $\mathsf{ping}{=}\varsigma(s)s$. Thus, $a.\mathsf{ping}$ returns the "identity" of an object o resulting from the evaluation of a; note that, due to aliasing and forwarding, this would be the "identity" of the current endpoint of an alias chain potentially starting at object o.

Self-Infliction, Serialization, Protection. Requests for operations on Øjeblik-objects may appear either (i) somewhere within a method body, or (ii) just within a let-body, or (iii) at top-level. The *current self of a request* denotes, in case (i), the self of its surrounding method declaration; in the other cases, it is undefined. A request for an Øjeblik operation is *self-inflicted*/*internal*, if it addresses its current self; otherwise, it is *external*. For instance, the term

$$[\,\mathrm{l}{=}\varsigma(s)s.\mathsf{clone}\,].\mathrm{l} \tag{1}$$

leads to an internal clone-request. However, not only literal invocations on the self variable s may be internal, but also indirect invocations on expressions that

evaluate to the object itself may be internal. For instance, also in

$$\mathsf{let}\, x = [\,\mathrm{l}{=}\varsigma(s,z)z.\mathsf{clone}\,]\,\mathsf{in}\, x.\mathrm{l}\langle x\rangle \tag{2}$$

the call $z.\mathsf{clone}$ will be internal when it is finally executed.

In concurrent object-based settings, the invariant that at most one thread at a time may be active within an object is called *serialization*. One way to ensure serialization is to associate *mutexes* with objects, which must be locked when a thread enters an object and released when the thread exits the object. In Obliq, the variant of *self-serialization* requires that the mutex is always acquired for external operations, but never for internal ones. For instance, the program

$$\mathsf{let}\, x = [\,\mathrm{l}{=}\varsigma(s)s.\mathrm{k}\,,\,\mathrm{k}{=}\varsigma(s)s\,]\,\mathsf{in}\, x.\mathrm{l}$$

will terminate (delivering as a result the identity of x), because the internal call to method k is permitted. In contrast, the program

$$\mathsf{let}\, x = [\,\mathrm{l}{=}\varsigma(s,z)z.\mathrm{k}\,,\,\mathrm{m}{=}\varsigma(s)s\,]\,\mathsf{in}\,\mathsf{let}\, y = [\,\mathrm{k}{=}\varsigma(s)x.\mathrm{m}\,]\,\mathsf{in}\, x.\mathrm{l}\langle y\rangle$$

attempts a mutual recursion between the objects x and y. However, it blocks the recursive (external) call from y to x for method m, because the mutex x is already locked by the former call of l on x, which has not yet terminated.

Øjeblik objects are *protected* against external modifications in a natural way: updates, cloning, and aliasing are only allowed if these operations are internal. For instance, the terms (1) and (2) terminate successfully (with a result), while

$$\mathsf{let}\, x = [\,\mathrm{l}{=}\varsigma(s)s\,]\,\mathsf{in}\, x.\mathsf{clone}$$

blocks (without result), because the clone-request is external.

In summary, operations on Øjeblik objects can be classified according to *protection conditions* and with respect to the *node of action* denoting the node where the operation is finally carried out (locally at the initially called node, or at the endpoint of a chain starting at the called node).

operation	*protection condition?*	*node of action*
cloning, aliasing	internal-only	local
update	internal-only	endpoint
invocation, surrogation, ping	unconstrained	endpoint

Scoping. Øjeblik offers scope declarations. An expression $\mathsf{let}\, x = a\,\mathsf{in}\, b$ first evaluates a, binding the result to x, and then evaluates b within the scope of the new binding. We use the standard inductive definition $\mathrm{fv}(a)$ to denote the free variables of term a with respect to method- and let-binding. Øjeblik only admits non-recursive expressions $\mathsf{let}\, x = a\,\mathsf{in}\, b$, i.e., with $x \notin \mathrm{fv}(a)$. Then, $a; b$ denotes $\mathsf{let}\, x = a\,\mathsf{in}\, b$, where $x \notin \mathrm{fv}(b)$. A term a is *closed* if $\mathrm{fv}(a) = \emptyset$.

$$a, b \quad ::= \quad \ldots \mid v \mid \mathsf{wait}$$

Figure 2. Syntax of Øjeblik run-time expressions

Concurrency. Computational activity takes place within *threads*. Apart from the main thread that is started on initialization, new separate threads can be created by the fork command. The term $\mathsf{fork}\langle a \rangle$ returns a new thread identifier to denote the thread evaluating a. The result of a fork'ed computation is grabbed by the join command. If a evaluates to a thread identifier, then $\mathsf{join}\langle a \rangle$ potentially blocks until that thread finishes and returns the thread's result, or blocks forever, if a join on thread a was already performed earlier.

2.2. Operational Semantics

The semantics performs local changes on global run-time *configurations*, which are mappings from references $v \in \mathbf{R}$ to run-time entities. More precisely, a configuration $\mathfrak{C}$ maps task references $t \in \mathbf{R}_{\mathcal{T}}$ to *tasks* $\mathcal{T}$, and object references $o \in \mathbf{R}_{\mathcal{O}}$ to *objects* $\mathcal{O}$ (see below). We use $\mathrm{dom}_X(\mathfrak{C})$ to denote $\mathrm{dom}(\mathfrak{C}) \cap \mathbf{R}_X$ for $X \in \{\mathcal{T}, \mathcal{O}\}$, and $\uparrow$ for undefined references.

Run-Time Entities. Run-time expressions a are generated from the extended Øjeblik grammar in Figure 2, where we introduce references v as *values*, as well as an additional construct wait whose meaning will become clear from its use later on. We refer to this extended set of terms as $\mathcal{L}_{\mathbf{R}}$. A run-time object $O \in \mathcal{O}$ is either an object record $\mathbb{O}$ (ranging over $[\mathrm{l}_j{=}m_j]_{j \in J}$) or a pointer $\gg o$ to an object reference $o \in \mathbf{R}_{\mathcal{O}}$. A run-time task T is a triple $\langle\, p, s, a \,\rangle \in \mathbf{R}_{\mathcal{T}} \times \mathbf{R}_{\mathcal{O}} \times \mathcal{L}_{\mathbf{R}}$ that refers to a *parent* p, a current *self* s, and a run-time expression a that remains to be evaluated. By the partial functions $\mathrm{s}_{\mathfrak{C}}(t)$ and $\mathrm{p}_{\mathfrak{C}}(t)$, we refer to the current self and parent of the task associated with reference t in $\mathfrak{C}$. We reserve the task references $t_{\mathrm{m}}, t_{\mathrm{g}} \in \mathbf{R}_{\mathcal{T}}$ for special purposes. In the following, we only consider *closed configurations*: every variable occurring in a run-time expression is bound within that expression, and every reference occurring in run-time expressions or in the codomain for object references is defined by the very configuration.

Alias chains. The partial function $\mathrm{ali}_{\mathfrak{C}} : \mathbf{R}_{\mathcal{O}} \rightharpoonup \mathbf{R}^*_{\mathcal{O}} \cup (\mathbf{R}^*_{\mathcal{O}} \cdot \{\uparrow\})$ with

$$\mathrm{ali}_{\mathfrak{C}}(o) \stackrel{\mathrm{def}}{=} \begin{cases} \uparrow & \text{if } \mathfrak{C}(o) = \uparrow \\ o & \text{if } \mathfrak{C}(o) = \mathbb{O} \\ o \cdot \mathrm{ali}_{\mathfrak{C}}(o') & \text{if } \mathfrak{C}(o) = \gg o' \end{cases}$$

computes the *alias chain*, starting at reference o, where $\cdot$ denotes concatenation of (sets of) strings of references, in general possibly ending with $\uparrow$. This

$$
\begin{array}{rcl}
r & ::= & \mathbb{O} \mid \mathsf{wait} \mid o.\mathrm{l}{\Leftarrow}m \mid o.\mathrm{l}\langle\, \tilde{v} \,\rangle \\
& \mid & o.\mathsf{clone} \mid o.\mathsf{alias}\langle o' \rangle \\
& \mid & o.\mathsf{surrogate} \mid o.\mathsf{ping} \\
& \mid & \mathsf{let}\, x = v\, \mathsf{in}\, b \mid \mathsf{fork}\langle a \rangle \mid \mathsf{join}\langle t \rangle \\
\\
e[\cdot] & ::= & [\cdot] \mid e[\cdot].\mathrm{l}{\Leftarrow}m \mid e[\cdot].\mathrm{l}\langle\, \tilde{a} \,\rangle \mid o.\mathrm{l}\langle\, \tilde{v}, e[\cdot], \tilde{a} \,\rangle \\
& \mid & e[\cdot].\mathsf{clone} \mid e[\cdot].\mathsf{alias}\langle b \rangle \mid o.\mathsf{alias}\langle e[\cdot] \rangle \\
& \mid & e[\cdot].\mathsf{surrogate} \mid e[\cdot].\mathsf{ping} \\
& \mid & \mathsf{let}\, x = e[\cdot]\, \mathsf{in}\, b \mid \mathsf{join}\langle e[\cdot] \rangle
\end{array}
$$

Figure 3. Evaluation of Øjeblik run-time expressions

computation only terminates, if there are no cycles in the chain. The endpoint of an alias chain is denoted by $\mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o))$; if it exists, then the semantics will guarantee that it is associated with an object record $\mathbb{O}$. We write $o' \in \mathrm{ali}_{\mathfrak{C}}(o)$ if o' occurs in the string representing the alias chain starting at o.

As a specialization of the above function, we define

$$
\mathrm{pre}_{\mathfrak{C}}(o, s) \stackrel{\mathrm{def}}{=} \begin{cases} \uparrow & \text{if } \mathfrak{C}(o) = \uparrow \\ o & \text{if } \mathfrak{C}(o) = \mathbb{O} \text{ or } o = s \\ o \cdot \mathrm{pre}_{\mathfrak{C}}(o', s) & \text{if } \mathfrak{C}(o) = {\gg}o' \text{ and } o \neq s \end{cases}
$$

which yields the prefix of the alias chain starting in o that ends with the first occurrence of s, if it exists. If $s \notin \mathrm{ali}_{\mathfrak{C}}(o)$, then $\mathrm{pre}_{\mathfrak{C}}(o, s) = \mathrm{ali}_{\mathfrak{C}}(o)$.

We sometimes refer to object references as *nodes*, reflecting the fact that they may denote nodes in an alias chain. A node $o \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$ is *active* if there is $t \in \mathrm{dom}_{\mathcal{T}}(\mathfrak{C})$ with $\mathrm{s}_{\mathfrak{C}}(t) = o$, otherwise it is called *idle*.

Evaluation. Figure 3 contains grammars to generate *redexes* r and *evaluation contexts* $e[\cdot]$ used to control the leftmost-innermost evaluation [FF86] of run-time expressions. A simple algorithm computes for every closed run-time expression $a \notin \mathbf{R}$ a *unique* pair of redex r and context $e[\cdot]$ such that $a = e[r]$.

Behaviors. The semantics of a closed term a is given by assigning to it the initial configuration $[\![\, a \,]\!] := \{t_{\mathrm{m}}{:=}\langle \uparrow, \uparrow, a \,\rangle, t_{\mathrm{g}}{:=}\langle \uparrow, \uparrow, t_{\mathrm{m}} \,\rangle\}$. The task referred to by t_{m} represents the start of the so-called *main* thread; the task reference t_{g} is used as the parent of all *garbage* task references, i.e., references that should not be reused, although their referred tasks are accomplished.

The behavior of configurations is generated from the syntax-directed transition rules in Figure 4. In each case we pick some task and object references in a particular configuration $\mathfrak{C}$, which under the respective conditions may enable a transition to take place in $\mathfrak{C}$. In the premises, note that the expressions of tasks are always in unique context-redex decomposed form. In the conclusions

$$\frac{\mathfrak{C}(t) = \langle\, p, s, e[\mathsf{let}\, x = v\, \mathsf{in}\, b]\,\rangle}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, p, s, e[b\{^{v}/_{x}\}]\,\rangle\}} \quad (\textsc{Let})$$

$$\frac{\mathfrak{C}(t) = \langle\, p, s, e[\mathbb{O}]\,\rangle \qquad \mathfrak{C}(o) = \uparrow}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, p, s, e[o]\,\rangle, o := \mathbb{O}\}} \quad (\textsc{New})$$

$$\frac{\mathfrak{C}(t) = \langle\, p, s, e[\mathsf{fork}\langle a\rangle]\,\rangle \qquad \mathfrak{C}(t') = \uparrow}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, p, s, e[t']\,\rangle, t' := \langle\, \uparrow, \uparrow, a\,\rangle\}} \quad (\textsc{Fork})$$

$$\frac{\mathfrak{C}(t) = \langle\, p, s, e[\mathsf{join}\langle t'\rangle]\,\rangle \qquad \mathfrak{C}(t') = \langle\, \uparrow, \uparrow, v\,\rangle}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, p, s, e[v]\,\rangle, t' := \langle\, t_{\mathrm{g}}, \uparrow, v\,\rangle\}} \quad (\textsc{Join})$$

$$\frac{\begin{array}{l}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathrm{l}_k\langle\, \tilde{v}\,\rangle]\,\rangle \qquad \mathfrak{C}(t') = \uparrow \\ \mathfrak{C}(\hat{o}) = [\mathrm{l}_j{=}\varsigma(s_j, \tilde{x}_j)b_j]_{j\in J} \qquad k{\in}J \\ \forall \dot{o} \in \mathrm{ali}_{\mathfrak{C}}(o) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t) \qquad \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o)) = \hat{o}\end{array}}{\mathfrak{C} \;\to \mathfrak{C}\{\, t := \langle\, p, s, e[\mathsf{wait}]\,\rangle, t' := \langle\, t, \hat{o}, b_k\{^{\tilde{o}\tilde{v}}/_{s_k\tilde{x}_k}\}\,\rangle\}} \quad (\textsc{Inv})$$

$$\frac{\mathfrak{C}(t) = \langle\, p, s, e[\mathsf{wait}]\,\rangle \qquad \mathfrak{C}(t') = \langle\, t, s', v\,\rangle}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, p, s, e[v]\,\rangle, t' := \langle\, t_{\mathrm{g}}, \uparrow, v\,\rangle\}} \quad (\textsc{Ret})$$

$$\frac{\begin{array}{l}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathrm{l}_k{\Leftarrow}m]\,\rangle \\ \mathfrak{C}(s) = [\mathrm{l}_j{=}m_j]_{j\in J} \qquad k{\in}J \\ \forall \dot{o} \in \mathrm{ali}_{\mathfrak{C}}(o) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t) \qquad \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o)) = s\end{array}}{\mathfrak{C} \;\to \mathfrak{C}\{\, t := \langle\, p, s, e[s]\,\rangle, s := [\mathrm{l}_k{=}m, \mathrm{l}_{j\neq k}{=}m_j]_{j\in J}\}} \quad (\textsc{Upd})$$

$$\frac{\begin{array}{l}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathsf{clone}]\,\rangle \qquad \mathfrak{C}(o') = \uparrow \\ \forall \dot{o} \in \mathrm{pre}_{\mathfrak{C}}(o, s) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t) \qquad s \in \mathrm{ali}_{\mathfrak{C}}(o)\end{array}}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, p, s, e[o']\,\rangle, o' := \mathfrak{C}(s)\}} \quad (\textsc{Cln})$$

$$\frac{\begin{array}{l}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathsf{alias}\langle o'\rangle]\,\rangle \\ \forall \dot{o} \in \mathrm{pre}_{\mathfrak{C}}(o, s) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t) \qquad s \in \mathrm{ali}_{\mathfrak{C}}(o)\end{array}}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, p, s, e[o']\,\rangle, s := \gg o'\}} \quad (\textsc{Ali})$$

Figure 4. Structural Operational Semantics

of the rules, $\mathfrak{C}\{t{:=}T, o{:=}O\}$ means that the mapping $\mathfrak{C}$ is either extended or overwritten with the association of task reference t with task T, and object reference o with run-time object O.

(LET) and (NEW) describe the local activity in a single task t in a straightforward manner; recall that let is not recursive. Furthermore, we assume that the value v is either a task or an object reference whose actual run-time entity is accessible through $\mathfrak{C}$. In rule (FORK), a new task t' is spawned off, which runs the expression a without current self. In rule (JOIN), the parent referring to its child t' is returned a value v. Note that fork'ed tasks do not know their parent, so they indeed represent initial tasks of new threads. As soon as a thread t is join'ed, it is marked as garbage by means of the special reference t_{g} as its parent; no further attempt to join t will succeed, and t can not be reused after the first join. (INV) and (RET) run a synchronous method invocation protocol. In (INV), a call to an object results in the creation of a new (callee-) task *within* the target object, while the caller-task is delayed, which is syntactically represented by the term wait inserted into its evaluation context. In rule (RET), this caller-callee pair can communicate the result as soon as the callee-expression has reduced to a value; the callee afterwards refers to the garbage reference. The rules (CLN)/(ALI)/(INV)/(UPD) crucially depend on the fact whether the alias chain—starting at the object on which the operation is requested—is "available" for this request. The idea is to check whether a request is allowed either to be performed in a node along the chain, as in rules (CLN)/(ALI) using the function $\mathrm{pre}_{\mathfrak{C}}(o, s)$, or to be passed on to the endpoint of the chain, as in rules (INV)/(UPD) using the function $\mathrm{ali}_{\mathfrak{C}}(o)$. An individual object o is *available* for task t in $\mathfrak{C}$, if o is idle, or if it is the same as the current self of t, such that operations from t on o would be internal:

$$\mathrm{Avail}_{\mathfrak{C}}(o,t) \stackrel{\mathrm{def}}{=} \underbrace{\bigwedge_{t' \in \mathrm{dom}_{\mathcal{T}}(\mathfrak{C})} (o \neq \mathrm{s}_{\mathfrak{C}}(t'))}_{o \text{ is idle}} \quad \vee \quad \underbrace{(o = \mathrm{s}_{\mathfrak{C}}(t))}_{\text{internal}}$$

Apart from availability, the rules (CLN)/(ALI)/(UPD) are completely straightforward according to the informal semantics explained in Subsection 2.1.

Both surrogate and ping are semantically regarded as standard methods, except that they are not updatable. Thus, the treatment of requests for surrogate and ping is analogous (INV), except that there is no requirement $k{\in}J$ to match one of the defined labels since surrogate and ping are implicitly present.

For convenience, we sometimes label transitions with task references. This provides precise information about the rule underlying it, because the run-time expression inhabiting a task is uniquely decomposed into redex and context. For example, $\mathfrak{C} \xrightarrow{t} \mathfrak{C}'$ denotes that the transition is derived by exploiting the run-time expression of task $\mathfrak{C}(t)$. $\mathfrak{C} \xrightarrow{t:(\mathrm{I})} \mathfrak{C}'$ in addition explicits that rule (INV) was employed for the derivation. (For more precision, one could even

add the freshly chosen names as additional labels.) Similarly, by $\mathfrak{C} \xrightarrow{\neg t} \mathfrak{C}'$ we schematically denote those transitions which do not touch the task at t.

2.3. Behavioral Semantics

We define contextual equivalences based on convergence [Mor68].

Definition 1 (Computation & Convergence) *Let $\mathfrak{C}$ be a configuration.*

1 *A* computation $\mathfrak{c}$ *(*starting at $\mathfrak{C}_0$*) is*

   (a) *either an infinite sequence $(\mathfrak{C}_i)_{0\leq i}$ of configurations with $\forall 0{\leq}i : \mathfrak{C}_i \ \rightarrow \mathfrak{C}_{i+1}$,*

   (b) *or a finite sequence $(\mathfrak{C}_i)_{0\leq i\leq n}$ of configurations with $\forall 0{\leq}i{<}n : \mathfrak{C}_i \ \rightarrow \mathfrak{C}_{i+1}$ and $\mathfrak{C}_n \not\rightarrow$.*

2 *Let $\mathfrak{c} := (\mathfrak{C}_i)_i$ be a computation starting at $\mathfrak{C}$. Then $\mathfrak{c}$ is called* successful, *written $\mathfrak{c}{\Downarrow}$, if there is $0{\leq}s$ and value v such that $\mathfrak{C}_s(t_{\mathrm{m}}) = \langle \uparrow, \uparrow, v \rangle$.*

3 (a) $\mathfrak{C}$ may converge, *written $\mathfrak{C}{\Downarrow}^{\mathrm{may}}$, if there is a successful computation starting at $\mathfrak{C}$.*

   (b) $\mathfrak{C}$ must converge, *written $\mathfrak{C}{\Downarrow}^{\mathrm{must}}$, if all computations starting at $\mathfrak{C}$ are successful.*

4 *Let a be a closed Øjeblik term. Then $a{\Downarrow}^{\mathrm{may}}$ if $[\![\, a \,]\!]{\Downarrow}^{\mathrm{may}}$, and $a{\Downarrow}^{\mathrm{must}}$ if $[\![\, a \,]\!]{\Downarrow}^{\mathrm{must}}$.*

This notion of success and convergence does not mean that the computation of term a terminates, but rather that the main task t_{m} does so. Note that there might be fork'ed tasks around that have not yet been join'ed, and which may possibly run forever.

An *Øjeblik program context* $C[\cdot]$ is an Øjeblik term with a single hole $[\cdot]$ that may be filled with an Øjeblik term; we omit the straightforward formal definition. A context $C[\cdot]$ is *closing* a term a, if $C[a]$ is closed.

Definition 2 *Let $a, b \in \mathcal{L}$ and $\mathbf{C}$ be a set of contexts closing a, b.*

1 *a and b are* may-equivalent *w.r.t. $\mathbf{C}$ written $a \cong_{\mathbf{C}}^{\mathrm{may}} b$, if for all $C[\cdot] \in \mathbf{C}$: $C[a]{\Downarrow}^{\mathrm{may}}$ iff $C[b]{\Downarrow}^{\mathrm{may}}$.*

2 *a and b are* must-equivalent *w.r.t. $\mathbf{C}$ written $a \cong_{\mathbf{C}}^{\mathrm{must}} b$, if for all $C[\cdot] \in \mathbf{C}$: $C[a]{\Downarrow}^{\mathrm{must}}$ iff $C[b]{\Downarrow}^{\mathrm{must}}$.*

In a typed language such as Øjeblik [NHKM01], it is natural to only consider well-typed terms, i.e., only contexts yielding well-typed composites. The results of the current paper are robust w.r.t. this adaptation.

3. On the Safety of Surrogation

In [NHKM01], we motivated an equation on Øjeblik terms to model the safety of object surrogation in the sense that object surrogation should be transparent to object clients. In other words, *an object should behave the same with or without surrogation* in all possible contexts (in $\mathbf{C}$).

$$x.\mathsf{ping} \quad \cong^{\mathrm{may}}_{\mathbf{C}} \quad x.\mathsf{surrogate}$$

One of the main observations in [NHKM01] was that the safety equation can not hold for all Øjeblik-contexts: problematic are those in which the operation $x.\mathsf{surrogate}$ could occur internally. The reason is that *internal* surrogation might lead to a misuse, by intention or by accident, of the newly created references. For example, let us look at the contexts

$$\begin{array}{lcl} C_1[\cdot] & \stackrel{\mathrm{def}}{=} & [\,\mathrm{l}{=}\varsigma(s)[\cdot].\mathsf{clone}\,].\mathrm{l} \\ C_2[\cdot] & \stackrel{\mathrm{def}}{=} & \mathsf{let}\, x = [\,\mathrm{l}{=}\varsigma(s,z)[\cdot].\mathsf{clone}\,]\,\mathsf{in}\, x.\mathrm{l}\langle x\rangle \end{array}$$

which perform a cloning operation on the hole inside a method. Note that the access to s from within the hole is internal. If we plug $s.\mathsf{surrogate}$ into the hole, then the cloning will be carried out on the result of the internal surrogate. However, since the surrogate returns a reference to the just created copy, the clone will be external and block. If we plug $s.\mathsf{ping}$ into the hole, then the cloning will be performed without problems: here, it is internal due to ping in this case returning just the current self of its surrounding method. We get:

$$C_i[s.\mathsf{surrogate}]\not\Downarrow \qquad \text{and} \qquad C_i[s.\mathsf{ping}]\Downarrow$$

In both cases, there are only deterministic reductions: in contrast to the case of surrogate, the case of ping leads to a successful final state.

In [NHKM01], we conjectured that in our semantics at least external surrogation is safe. To deal with the undecidable criterion of external requests [Car95] (hinted at by the above example), we introduced "tagged" requests as additional versions of surrogation and ping. Tagging helps us to detect all "requests arising from the hole", i.e., if we start the evaluation of a context with a tagged subterm plugged in, then we may check at any time whether, in a run-time expression, a tagged subterm appears as top-level redex.

Definition 3 *Let $C[\cdot]$ be a context with $C[x]$ closed. Then, $C[\cdot]$ is called* external *for x if $[\![\, C[x.\mathsf{ping}^\star] \,]\!] \;\rightarrow^*\; \mathfrak{C}$ with $\mathfrak{C}(t) = \langle\, p, s, e[o.\mathsf{ping}^\star] \,\rangle$ and $\forall \dot{o} \in \mathrm{ali}_{\mathfrak{C}}(o) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t)$ implies $s \neq \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o))$.*

We let $\mathbf{E}(x)$ denote the set of Øjeblik contexts external for x.

Theorem 1 (Safety) *Let x be a variable. Let $\mathrm{m} \in \{\mathrm{may}, \mathrm{must}\}$. Then:*

$$x.\mathsf{ping} \quad \cong^{\mathrm{m}}_{\mathbf{E}(x)} \quad x.\mathsf{surrogate}.$$

In [MKN00], we indeed proved a variant of Theorem 1 based on π-calculus notions of may-convergence and -equivalence *for translations* of Øjeblik-terms. In the next section, we summarize a new proof, now based on the operational semantics of Øjeblik terms *themselves*. Moreover, we prove Theorem 1 for must-equivalence, which was not treated in previous work because of the insensitiveness of the standard bisimulation proof techniques w.r.t. divergence, which matters for must-equivalence.

4. Proving Safety

Proving Theorem 1 amounts to the mutual simulation of computations starting in $C[x.\mathsf{ping}^\star]$ and $C[x.\mathsf{surrogate}^\star]$. Here, we exemplify the proof for must-equivalence: $a \cong^{\mathrm{must}}_{\mathbf{E}(x)} b$ requires us to prove that $C[a]\Downarrow^{\mathrm{must}}$ iff $C[b]\Downarrow^{\mathrm{must}}$ for all $C[\cdot] \in \mathbf{E}(x)$. The direct proof requires the exhibition of success for an infinite number of computations for each context. Instead, we choose the equivalent formulation that requires us to prove that ($\exists \mathfrak{p}$ starting at $C[x.\mathsf{ping}^\star]$ with $\neg\mathfrak{p}\Downarrow$) iff ($\exists \mathfrak{s}$ starting at $C[x.\mathsf{surrogate}^\star]$ with $\neg\mathfrak{s}\Downarrow$). Summing up, for must-equivalence we have to simulate unsuccessful computations. In contrast, for may-equivalence we would have to simulate successful ones [MKN00].

4.1. Overview

We borrow from the strategy used in [MKN00] and distinguish among the transitions occurring in computations *significant* from *insignificant* ones.

Definition 4 (Significant transitions)
Let $(\mathfrak{C}_i)_{0\leq i}$ *be a finite or infinite computation starting at* $\mathfrak{C}_0 = C[x.\mathsf{op}^\star]$.
A transition $\mathfrak{C}_i \xrightarrow{t_i} \mathfrak{C}_{i+1}$ *is* significant, *if* $\mathfrak{C}_i(t_i) = \langle\, p, s, e[o.\mathsf{op}^\star] \,\rangle$.

Every transition that represents the invocation of a tagged request is significant, because only such transitions may cause different behaviors; every other transition is contributed by the program context and can thus be simulated trivially.

PROOF. [of $x.\mathsf{ping} \cong^{\mathrm{must}}_{\mathbf{E}(x)} x.\mathsf{surrogate}$] (Full proof in [Bri01]).

In Figure 5, we sketch the constructive simulation of a computation starting at $C[x.\mathsf{ping}^\star]$ by a computation starting at $C[x.\mathsf{surrogate}^\star]$. We denote the significant transitions by $\rightarrow_{\mathrm{s}}$, so $\mathfrak{C}_{i,m_i}(t_i) = \langle\, p_i, s_i, e_i[o_i.\mathsf{ping}^\star] \,\rangle$. By the syntactic relabeling function $[{}^{\mathsf{surrogate}^\star}/_{\mathsf{ping}^\star}]$, we define:

$$\forall 1 \leq i, \forall 1 \leq j \leq m_i, \mathfrak{C}'_{i,j} \stackrel{\mathrm{def}}{=} \mathfrak{C}_{i,j}[{}^{\mathsf{surrogate}^\star}/_{\mathsf{ping}^\star}]$$

Note that, by this construction, a $\mathsf{ping}^\star$ enabled in $\mathfrak{C}_{i,j}$ implies that a $\mathsf{surrogate}^\star$ is enabled in $\mathfrak{C}'_{i,j}$. So, whenever a significant $\mathsf{ping}^\star$ needs to be simulated, we invoke the respective $\mathsf{surrogate}^\star$ and immediately perform the cloning and

Let $\exists \mathfrak{p}$ starting at $\mathfrak{C}_{1,1} \stackrel{\text{def}}{=} C[x.\mathsf{ping}^\star]$ with $\neg \mathfrak{p}\Downarrow$:

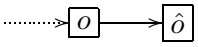
$$\begin{array}{lllll}
\mathfrak{C}_{1,1} & \to \cdots \to & \mathfrak{C}_{1,m_1} & \xrightarrow{t_1:(\mathrm{I})}_{\mathrm{s}} & \mathfrak{C}_1 = \mathfrak{C}_{2,1} \\
\mathfrak{C}_{2,1} & \to \cdots \to & \mathfrak{C}_{2,m_2} & \xrightarrow{t_2:(\mathrm{I})}_{\mathrm{s}} & \mathfrak{C}_2 = \mathfrak{C}_{3,1} \\
\vdots & & \vdots & & \vdots \\
\mathfrak{C}_{n,1} & \to \cdots \to & \mathfrak{C}_{n,m_n} & \xrightarrow{t_n:(\mathrm{I})}_{\mathrm{s}} & \mathfrak{C}_n = \mathfrak{C}_{n+1,1} \\
\vdots & & \vdots & & \vdots
\end{array}$$

Then $\exists \mathfrak{s}$ starting at $\mathfrak{C}'_{1,1} \stackrel{\text{def}}{=} C[x.\mathsf{surrogate}^\star]$ with $\neg \mathfrak{s}\Downarrow$:

$$\begin{array}{lllllll}
\mathfrak{C}'_{1,1} & \to \cdots \to & \mathfrak{C}'_{1,m_1} & \xrightarrow{t_1:(\mathrm{I})}_{\mathrm{s}} & \mathfrak{C}'_1 \xrightarrow{t_1:(\mathrm{C})} \xrightarrow{t_1:(\mathrm{A})} & \widehat{\mathfrak{C}}_1 & \cong^{\mathrm{must}}_{\mathbf{E}(x)} \ \mathfrak{C}'_{2,1} \\
\mathfrak{C}'_{2,1} & \to \cdots \to & \mathfrak{C}'_{2,m_2} & \xrightarrow{t_2:(\mathrm{I})}_{\mathrm{s}} & \mathfrak{C}'_2 \xrightarrow{t_2:(\mathrm{C})} \xrightarrow{t_2:(\mathrm{A})} & \widehat{\mathfrak{C}}_2 & \cong^{\mathrm{must}}_{\mathbf{E}(x)} \ \mathfrak{C}'_{3,1} \\
\vdots & & \vdots & & \vdots & & \\
\mathfrak{C}'_{n,1} & \to \cdots \to & \mathfrak{C}'_{n,m_n} & \xrightarrow{t_n:(\mathrm{I})}_{\mathrm{s}} & \mathfrak{C}'_n \xrightarrow{t_n:(\mathrm{C})} \xrightarrow{t_n:(\mathrm{A})} & \widehat{\mathfrak{C}}_n & \cong^{\mathrm{must}}_{\mathbf{E}(x)} \ \mathfrak{C}'_{n+1,1} \\
\vdots & & \vdots & & \vdots & &
\end{array}$$

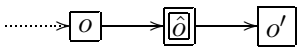
Figure 5. Simulating $\mathsf{ping}^\star$-Computations

aliasing. The configurations $\mathfrak{C}_i$ (resulting from $\mathsf{ping}^\star$) and $\widehat{\mathfrak{C}}_i$ (resulting from $\mathsf{surrogate}^\star$) are quite different: while the effect of $\mathsf{ping}^\star$ on an alias chain



(3)

ending in $\hat{o}$ is vacuous (it just returns $\hat{o}$), a $\mathsf{surrogate}^\star$ turns this chain into



(4)

in which $\hat{o}$ is a **stable alias**, which will never ever change again (cf. § 4.2). Since any incoming request will be forwarded to its successor o', we may as well direct all these requests directly to the successor: we call **path compression** the technique of manipulating a configuration through the elimination of stable aliases (cf. § 4.3). The proof of $\widehat{\mathfrak{C}}_i \cong^{\mathrm{must}}_{\mathbf{E}(x)} \mathfrak{C}_i$ works by manipulation of $\widehat{\mathfrak{C}}_i$ using this technique, while preserving and reflecting may- and must-convergence properties. Intuitively, in this proof, path compression allows us to "semantically undo" the effect of surrogation on configurations, such that the simulation $\mathfrak{s}$ can afterwards proceed again in lock-step with computation $\mathfrak{p}$.

In Figure 6, the converse is depicted. The significant transitions are now due to $\mathfrak{C}_{i,m_i}(t_i) = \langle\, p_i, s_i, e_i[o_i.\mathsf{surrogate}^\star] \,\rangle$. Yet, the simulation problem is considerably more difficult than in the case of $\mathsf{ping}^\star$, because the significant

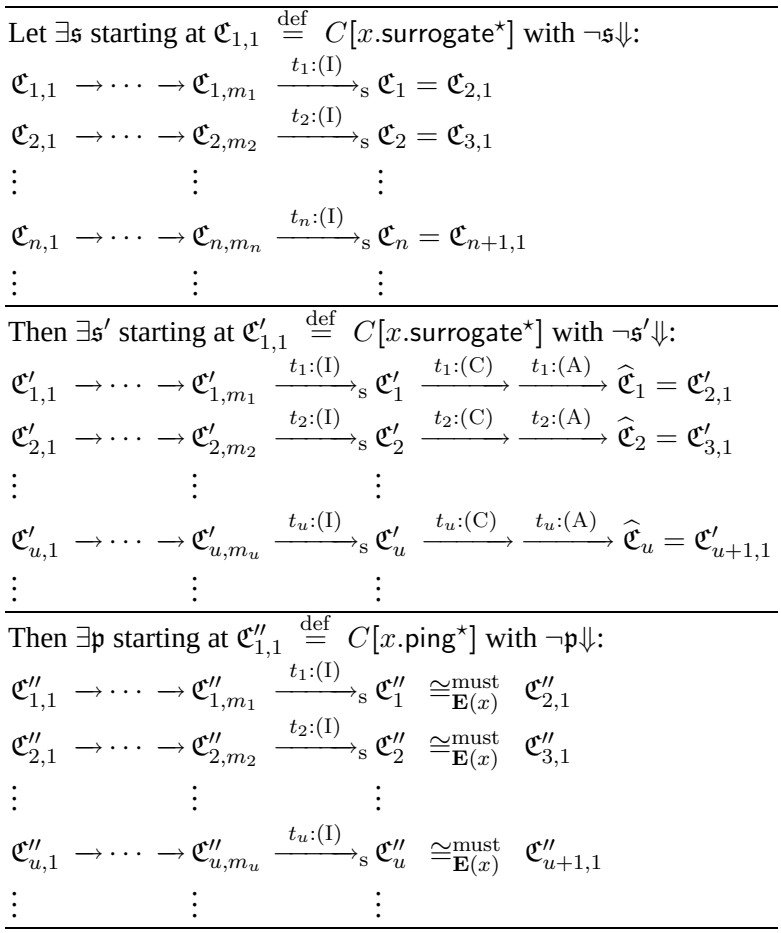


Figure 6. Simulating $\mathsf{surrogate}^\star$-Computations

steps calling $\mathsf{surrogate}^\star$ are not necessarily directly followed by the cloning and aliasing that required to complete surrogation. In a concurrent environment the completion might even be delayed arbitrarily. Therefore, we study **partial confluence** properties (cf. § 4.4), which allow us to reshuffle arbitrary computations $\mathfrak{s}$ so as to perform the required operations immediately, while preserving and reflecting the intended convergence behavior. Caution is due: in infinite computations, not every call of $\mathsf{surrogate}^\star$ must be completed. However, incomplete surrogations cannot have had an impact on the failure of the computation $\mathfrak{s}$, so we may either omit or complete those uncompleted significant steps in order to match the format of $\mathfrak{s}'$ in Figure 6. We then define:

$$\forall 1 \leq i, \forall 1 \leq j \leq m_i, \mathfrak{C}''_{i,j} \stackrel{\text{def}}{=} \mathfrak{C}'_{i,j}[{}^{\mathsf{ping}^\star}/_{\mathsf{surrogate}^\star}]$$

Now, analogous to the simulation of Figure 5, this time $\widehat{\mathfrak{C}}_i$ and $\mathfrak{C}_i''$ need to be related. Again, path compression on stable aliases in $\widehat{\mathfrak{C}}_i$ does the job. □

4.2. Stable Aliases

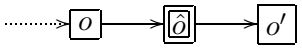
An alias node is a node $o \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$ with $\mathfrak{C}(o) = \gg o'$ for $o' \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$. An alias node $o \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$ is *stable*, if $\mathfrak{C} \rightarrow^* \mathfrak{C}'$ implies $\mathfrak{C}'(o) = \mathfrak{C}(o)$. Note that idle alias nodes are always stable. However, inactivity is not a necessary condition; any alias whose inhabiting task has reduced to a value is also stable.

Lemma 5 *Let o be an alias in $\mathfrak{C}$. Let $t \in \mathrm{dom}_{\mathcal{T}}(\mathfrak{C})$ and v be a value such that $\mathfrak{C}(t) = \langle\, p, o, v \,\rangle$ and for all $t' \neq t$: $\mathrm{s}_{\mathfrak{C}}(t') \neq o$. Then o is a stable alias in $\mathfrak{C}$.*

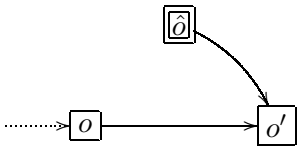
Of course, also this lemma does not represent a necessary condition, but it is sufficient for our proofs. Note that the result of surrogate* methods are tasks of precisely the form $\langle\, p, o, o' \,\rangle$ with o turned into $\gg o'$, so such o are stable.

4.3. Path Compression

The aim is to eliminate stable aliases, as the one displayed in (4), and to perform some convenient renaming afterwards in order to arrive at a situation as displayed in (3). To be useful, all of these manipulations must not affect the convergence properties of a configuration. The first step is *path compression*, which is a function $\mathrm{comp}_{\hat{o}}(\mathfrak{C})$, which replaces in configuration $\mathfrak{C}$ containing (4)

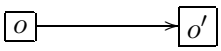


all references to $\hat{o}$, wherever they might occur in run-time expressions, as current self, or in aliases of the configuration, by o', i.e., the successor of $\hat{o}$ in $\mathfrak{C}$.



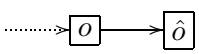
(5)

As a result of path compression, the reference $\hat{o}$ itself is now "unused". Consequently, a simple destructive function $\mathrm{elim}_{\hat{o}}(\cdot)$ may eliminate it.



(6)

Finally, another function $\mathrm{ren}_{\{o' \mapsto \hat{o}\}}(\cdot)$ performs the renaming of o' to $\hat{o}$, which provides us with a configuration



that relates directly to the configuration containing (3), i.e., the result of ping.

The crux of the "compress-eliminate-rename" procedure is that, properly defined, all three operations are indifferent w.r.t. convergence.

Lemma 6 *Let $\hat{o}$ be stable in $\mathfrak{C}$ with $\mathfrak{C}(\hat{o}) = \gg o'$. Let* $\mathrm{m}\in\{\mathrm{may}, \mathrm{must}\}$. *Then:*

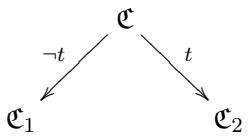
$$\mathfrak{C}\Downarrow^{\mathrm{m}} \textit{ iff } \quad \mathrm{ren}_{\{o'\mapsto\hat{o}\}}(\mathrm{elim}_{\hat{o}}(\mathrm{comp}_{\hat{o}}(\mathfrak{C})))\Downarrow^{\mathrm{m}}.$$

The detailed function definitions and proofs can be found in [Bri01].

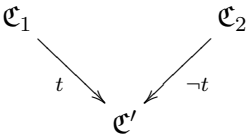
4.4. Confluence

The method $\mathsf{surrogate}{=}\varsigma(s)s.\mathsf{alias}\langle s.\mathsf{clone}\rangle$, once invoked, involves three transitions for cloning, aliasing, and returning its result. As a matter of fact, these transitions can not be preempted in finite computations by any other operation enabled at the same time. This fact is conveniently formalized as a confluence property, which we list here for the case of cloning and aliasing. (Confluence is of course not a new notion as such; it has been known in operational semantics and term rewriting for a long time. See [MT99] for an application in the context of semantics for Actor languages.)

Lemma 7 *Let $\mathfrak{C}$ be a configuration. Let $t_{\mathrm{m}} \neq t \in \mathbf{R}_{\mathcal{T}}$ and let $o \in \mathbf{R}_{\mathcal{O}}$ with $\mathfrak{C}(t) = \langle\, p, o, o.\mathsf{x}\,\rangle$ where $o.\mathsf{x}$ is a redex and for all $t' \neq t$: $\mathrm{s}_{\mathfrak{C}}(t') \neq o$. Let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be configurations with transitions*



where the transition labeled with $\neg t$ implies $\mathfrak{C}_1(t) = \mathfrak{C}(t)$ and $\mathfrak{C}_1(o) = \mathfrak{C}(o)$ as well as for all $t' \neq t$: $\mathrm{s}_{\mathfrak{C}_1}(t') \neq o$. Then there are



with $\mathfrak{C}'$ uniquely defined (up to the choice of fresh references):

1 *If* $\mathsf{x} = o.\mathsf{alias}\langle o.\mathsf{clone}\rangle$ *and* $o' \notin \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$,
then $\mathfrak{C}' \stackrel{\mathrm{def}}{=} \mathfrak{C}_1\{t := \langle\, p, o, o.\mathsf{alias}\langle o'\rangle\,\rangle, o' := \mathfrak{C}o\}$.

2 *If* $\mathsf{x} = o.\mathsf{alias}\langle o'\rangle$ *for* $o' \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$,
then $\mathfrak{C}' \stackrel{\mathrm{def}}{=} \mathfrak{C}_1\{t := \langle\, p, o, o'\,\rangle, o := \gg o'\}$.

PROOF. By case analysis on the enabled transitions ($\neg t$). □

As a consequence of the confluence lemma, we can exhibit that in any computation that enables the above operations of interest, these operations can be assumed to be carried out immediately. Moreover, such a manipulation of computations leaves unchanged the notion of success. Note that if a particular computation does not carry out an enabled operation, it must be infinite; otherwise, it could be extended by finally performing the enabled transition.

A further consequence of the confluence lemma is that the transitions that perform the interesting cloning and aliasing operations preserve and reflect both the may- and must-convergence behavior.

Lemma 8 *Let $\mathfrak{C}$ be a configuration. Let $t_{\mathrm{m}} \neq t \in \mathbf{R}_{\mathcal{T}}$ and let $o \in \mathbf{R}_{\mathcal{O}}$ with $\mathfrak{C}(t) = \langle\, p, o, o.\mathrm{x} \,\rangle$ where $o.\mathrm{x}$ is a redex and for all $t' \neq t : \mathrm{s}_{\mathfrak{C}}(t') \neq o$. Let $\mathfrak{C} \xrightarrow{t} \mathfrak{C}'$. Let $\mathrm{m} \in \{\mathrm{may}, \mathrm{must}\}$. Then $\mathfrak{C}{\Downarrow^{\mathrm{m}}}$ iff $\mathfrak{C}'{\Downarrow^{\mathrm{m}}}$.*

PROOF. By "chasing diagrams" and pasting them together. □

While there is also a confluence property (cf. Lemma 7) for the case of enabled (RET)-transitions involving task t, the respective Lemma 8 would not hold. Assume a surrogate operation that was called from within the main thread as its last operation. Obviously, performing the (RET)-transition yields success of the computation. Yet, there might be another task running an infinite loop, so there might be infinite computations in which success is never reached.

5. Conclusion

In this paper, we have sketched a proof of the safety of object surrogation (abstract object migration) using the operational semantics of Øjeblik. In addition to may-equivalence, which we had already shown in previously using a translational semantics, here we also prove the safety with respect to must-equivalence. The combination of the two results is powerful. Contexts that allow only successful computations with a surrogated object do so—by must-equivalence—if and only if they allow only successful computations with the unsurrogated object. Should there be unsuccessful computations (possibility of deadlock/divergence) allowed by some context enclosing a surrogated object, then—again by must-equivalence—the context will also allow for unsuccessful computations when enclosing the unsurrogated object. In addition, may-equivalence guarantees that surrogation does not add the possibility of success in case there is none for the unsurrogated object, nor does it remove the possibility of success in case there was one for the unsurrogated object. In summary, object surrogation does neither add or remove the possibility of success, nor does it add or remove the possibility of deadlock/divergence.

This paper underlines the conclusion of our whole project on the calculus Øjeblik: there are both pros and cons for either the translational semantics [MKN00] or the operational semantics [NHKM01]. The former is equipped

with a huge set of proof tools, allows us to study parts of concurrent programs separately and to discuss the design of the language implementation, but it lacks support for divergence-sensitive studies. The latter needs to be equipped with proper proof techniques from scratch, and it requires to study programs as a whole, but it and its proofs are generally easier to understand.

References

[Bri01] S. Briais. Banc d'essai de Funnel — Migration d'objets dans Øjeblik. Internship report, ENS Lyon, EPF Lausanne, Sept. 2001. In French. Available from `http://www.cs.auc.dk/research/FS/ojeblik/`.

[Car95] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995. Short version in *Proceedings of POPL '95*. A preliminary version appeared as Report 122, Digital Systems Research, June 1994.

[FF86] M. Felleisen and D. P. Friedman. Control Operators, the SECD-machine, and the λ-calculus. In M. Wirsing, ed, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[Mer00] M. Merro. *Locality in the π-calculus and applications to distributed objects*. PhD thesis, Ecole des Mines, France, October 2000.

[MKN00] M. Merro, J. Kleist and U. Nestmann. Local π-Calculus at Work: Mobile Objects as Mobile Processes. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses and T. Ito, eds, *Proceedings of TCS 2000*, volume 1872 of *LNCS*, pages 390–408. IFIP, Springer, Aug. 2000. Available from `http://www.cs.auc.dk/research/FS/ojeblik/`. Full version accepted for publication in *Journal of Information and Computation*.

[Mor68] J.-H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[MT99] I. A. Mason and C. L. Talcott. Actor Languages: Their Syntax, Semantics, Translation, and Equivalence. *Theoretical Computer Science*, 220:409 – 467, 1999.

[NHKM01] U. Nestmann, H. Hüttel, J. Kleist and M. Merro. Aliasing Models for Mobile Objects. Accepted for *Journal of Information and Computation*. Available from `http://www.cs.auc.dk/research/FS/ojeblik/`. An extended abstract has appeared as Distinguished Paper in the *Proceedings of EUROPAR '99*, LNCS 1685, 2001.